ABSTRACT:
A Network-Aware Distributed Storage Cache for Data Intensive Environments

Brian Tierney, Jason Lee, William Johnston, Brian Crowley, Mason Holding

Computing Sciences Division
Lawrence Berkeley National Laboratory[1]
University of California, Berkeley, CA, 94720

1.0 Introduction

High-speed data streams resulting from the operation of on-line instruments and imaging systems are a staple of modern scientific, health care, and intelligence environments. The advent of high-speed networks is providing the potential for new approaches to the collection, organization, storage, analysis, and distribution of the large-data-objects that result from such data streams. The result will be to make both the data and its analysis much more readily available.

For example, health care imaging systems illustrate the need for both high data rates and real-time cataloging. Medical video and image data used for diagnostic purposes — e.g., X-ray CT, MRI, and cardio-angiography — are collected at centralized facilities and may be accessed at locations other than the point of collection (e.g., the hospitals of the referring physicians). A second example is high energy physics experiments, which illustrate high rates and massive volumes of data that must be processed and archived in real time. This data must also be accessible to large scientific collaborations — typically hundreds of investigators at dozens of institutions around the world.

In this paper we will describe how “Computational Grid” environments can be used to help with these types of applications, and how a high-speed network cache is a particularly important component in a data intensive grid architecture. We describe our implementation of a network cache, how we have made it “network aware”, and how we do dynamic load balancing based on the current network conditions.

2.0 Data Intensive Grids

The integration of the various technological approaches being used to address the problem of integrated use of dispersed resources is frequently called a “grid,” or a computational grid - a name arising by analogy with the grid that supplies ubiquitous access to electric power. See, e.g., [7]. Basic grid services are those that locate, allocate, coordinate, utilize, and provide for human interaction with the various resources that actually perform useful functions.

Grids are built from collections of primarily independent services. The essential aspect of grid services is that they are uniformly available throughout the distributed environment of the Grid. Services may be grouped into integrated sets of services, sometimes called “middleware”. Current

1. The work described in this paper is supported by DARPA, Information Technology Office (http://www.darpa.mil/ito/ResearchAreas.html) and the U. S. Dept. of Energy, Office of Science, Office of Computational and Technology Research, Mathematical, Information, and Computational Sciences Division (http://www.er.doe.gov/production/octr/mics/index.html), under contract DE-AC03-76SF00098 with the University of California. This is report no. LBNL-NNNNN.

Grid tools include Globus [6], Legion [13], SRB [2], and workbench systems like Habanero [8] and WebFlow [1].

From the application's point of view, the Grid is a collection of middleware services that provide applications with a uniform view of distributed resource components and the mechanisms for assembling them into systems. From the middleware systems points of view, the Grid is a standardized set of basic services providing scheduling, resource discovery, global data directories, security, communication services, etc. However, from the Grid implementors point of view, these services result from and must interact with heterogeneous set of capabilities, and frequently involve "drilling" down through the various layers of the computing and communications infrastructure.

2.1 A Model Architecture for Data Intensive Environments

Our model is to use a high-speed distributed cache as a common element for all of the sources and sinks of data involved in high-performance data systems. This cache-based approach provides standard interfaces to a large, application-oriented, distributed, on-line, transient storage system. Each data source deposits its data in the cache, and each data consumer takes data from the cache, usually writing the processed data back to the cache. A tertiary storage system manager migrates data to and from the cache at various stages of processing. (See Figure 1.) We have used this model for data handling systems for high energy physics data and for medical imaging data. For more information see [12] and [11].

The high-speed cache serves several roles in this environment. It provides a standard high data rate interface for high-speed access by data sources, processing resources, mass storage systems (MSS), and user interface elements. It provides the functionality of a single very large, random access, block-oriented I/O device (i.e., a "virtual disk"). It serves to isolate the application from tertiary storage systems and instrument data sources. The cache also provides an "impedance matching" function between the coarse-grained nature of parallel tape drives in the tertiary storage system and the fine-grained access of hundreds of applications.

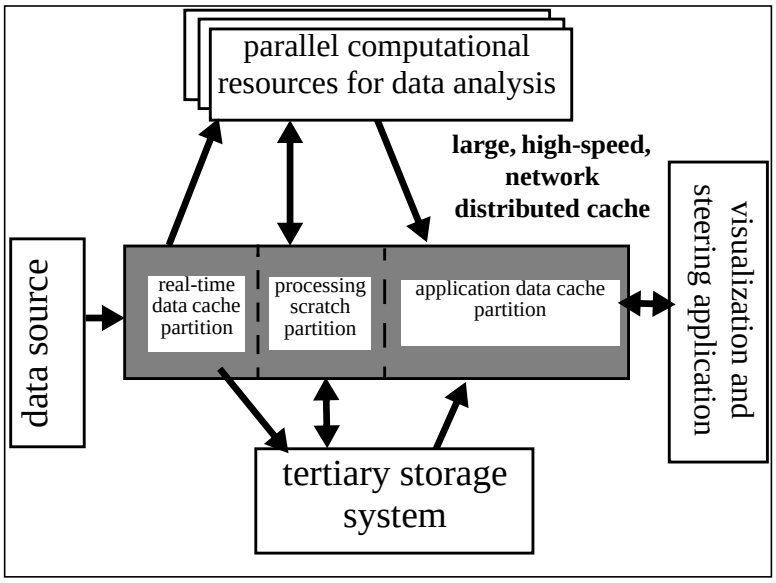

Figure 1: The Data Handling Model

Depending on the size of the cache relative to the objects of interest, the tertiary storage system management may only involve moving partial objects to the cache. In other words, the cache may contain a moving window for an extremely large off-line object/data set. Generally, the cache storage configuration is large compared to the available disks of a typical computing environment, and very large compared to any single disk (e.g. hundreds of gigabytes).

3.0 The Distributed-Parallel Storage System (DPSS)

Our implementation of this high-speed, distributed cache is called the Distributed-Parallel Storage System (DPSS) [18][5]. LBNL designed and implemented the DPSS as part of the DARPA MAGIC project [14], and as part of the U.S. Department of Energy's high-speed distributed computing program. This technology has been quite successful in providing an economical, high-performance, widely distributed, and highly scalable architecture for caching large amounts of data that can potentially be used by many different users.

The DPSS provides several important and unique capabilities for data intensive distributed computing environments. It provides application-specific interfaces to an extremely large space of logical blocks; it offers the ability to build large, high-performance storage systems from inexpensive commodity components; and it offers the ability to increase performance by increasing the number of parallel disk servers.

The application interface to the DPSS cache supports a variety of I/O semantics, including Unix-like I/O semantics through an easy to use client API library (e.g.: dpssOpen(), dpssRead(), dpssWrite(), dpssLSeek(), dpssClose()). The data layout on the disks is completely up to the application, and the usual strategy for sequential reading applications is to write the data "round-robin", striping blocks of data across the servers. The client library also includes a flexible data replication ability, allowing for multiple levels of fault tolerance. The DPSS client API is multi-threaded, where the number of client threads is equal to the number of DPSS servers. Therefore the speed of the client scales with the speed of the server, assuming the client host is powerful enough.

DPSS data blocks are available to clients immediately as they are placed into the cache. It it not necessary to wait until the entire file has been transferred before requesting data. This is particularly useful to clients requesting data from a tape archive. As the file moves from tape to the DPSS cache, the blocks in the cache are immediately available to the client. If a block is not available, the application can either block, waiting for the data to arrive, or continue to request other blocks of data which may be ready to read.

The internal architecture of the DPSS is illustrated in Figure 2. Requests for blocks of data are sent from the client to the "DPSS master" process, which determines which "DPSS block server" the blocks are located on, and forwards the requests to the appropriate server. The server then sends the block directly back to the client. Servers may be anywhere in the network: there is no assumption that they are all at the same location, or even the same city.

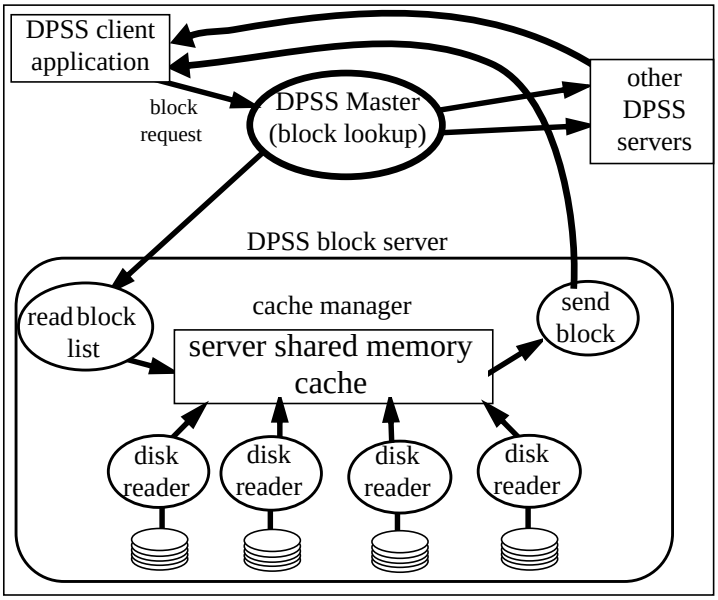

Figure 2: DPSS Architecture

Typical DPSS implementations consist of several low-cost workstations as DPSS block servers, each with several disk controllers, and several disks on each controller. A four-server DPSS can thus produce throughputs of over 50 MBytes/sec by providing parallel access to 20-30 disks.

The DPSS is dynamically reconfigurable, allowing one to add or remove servers or disks on-the-fly. This is done by storing the DPSS hardware resource information in a Globus Metacomputing Directory Service (MDS)[3] formatted LDAP database, which may be updated dynamically. Software agents are used to monitor network, host, and disk availability and load, storing this information into the LDAP database as well. This information can then be used for fault tolerance and load balancing. We describe this load balancing facility in more detail below.

4.0 Network-Aware Adaptation

For the DPSS cache to be effective in a wide area network environment, it must have sufficient knowledge of the network to adjust for a wide range of network performance conditions, and have sufficient adaptability to be able to dynamically reconfigure itself in the face of congestion and component failure.

4.1 Monitoring System

We have developed a software agent architecture for distributed system monitoring and management [10]. The agents, whose implementation is based on Java and RMI, can be used to launch a wide range of system and network monitoring tools, extract their results, and publish them into an LDAP database. We are currently using the agents to run *netperf* [15] and *ping* for network monitoring, and to *vmstat* and *uptime* for host monitoring. These results are uploaded to an LDAP database at regular intervals, typically every minute, for easy access by any process in the system. We run these agents on every host in the system, including the client host, so that we can learn about the network path between the client, the DPSS master, and all DPSS servers.

4.2 TCP Receive Buffers

The DPSS uses the TCP protocol for data transfers. For TCP to perform well over high-speeds networks, it is critical that there be enough buffer space for the congestion control algorithms work correctly [9]. Proper buffer size is a function of the network bandwidth-delay product, but because bandwidth-delay products in the Internet can span 4-5 orders of magnitude, it is impossible to configure the default TCP parameters on a host to be optimal for all connections. [16]

To solve this problem, the DPSS client library automatically determines the bandwidth-delay product for each connection to a DPSS server, and sets the TCP buffer size to the optimal value. The delay is measured by sending a few packets to the “echo” port on the DPSS server host. The bandwidth is obtained by monitoring the network connections using the monitoring agents described above, and requesting the current bandwidth from the LDAP database.

4.3 Load Balancing

Using the status information in the LDAP database, the DPSS master can easily find out the status of all the server hosts and network paths. Assuming the data blocks are replicated, the DPSS can now perform dynamic load balancing. There are several ways to compute the optimal distribution of block requests among a set of DPSS servers, and we are currently experimenting to try to determine the best method. The parameters to the load balancing algorithm are current network throughput and latency, and server CPU power, which is a function of which type of CPU that host has, how many CPU’s the host has, and what the current system load is. Tests performed varying only one of these parameters

shows that each affects the total system throughput linearly. Therefore we can use the following simple formula for doing load balancing:

server performance = (network throughput) x (CPU factor) x (Latency factor)

Where CPU factor = CPU strength x (1 - system load)

and Latency factor = (max latency - server latency) / (max latency - min latency)

We point out that there are several issues involved in obtaining accurate network throughput and latency measures. These issues will be discussed in the full version of this paper.

The DPSS master computes server performance of each server every time a list of blocks arrive, and sends a percentage of the requests to each server based on server performance. A typical DPSS client requests about 200 blocks at a time, and requests new group of blocks every few seconds. This usage model makes it easy for the DPSS to continually adjust which servers it is using to current network and host load.

5.0 Results

5.1 TCP Buffer Tuning

Table 1 shows the results from dynamic setting of the TCP receive buffer size. From this table one can see that by hand you can tune for either LAN access or WAN access, but not both at once. It is also apparent that while it is particularly important to set the buffer size big enough for the WAN case, it is also important not to set it too big for the LAN environment.

Table 1

buffer method	network	Total Throughput
hand tune for LAN (64KB buffers)	LAN	33 MBytes/sec
	WAN	5.5 MBytes/sec
hand tune for WAN (512 KB buffers)	LAN	19 MBytes/sec
	WAN	14 MBytes/sec
auto tune in DPSS library	LAN	33 MBytes/sec
	WAN	14 MBytes/sec
LAN RTT = 1 ms over OC-12 (622 Mbit/sec) network WAN RTT = 44 ms over a OC-3 (155 Mbit/sec) network		

5.2 Load Balancing

Figure 3, Figure 4, and Figure 5 show the results of using dynamic load balancing. In Figure 3 we used servers with the same load and latency, and varied the available network throughput (the first two servers were on OC-3, the third on 10BT ethernet, and the fourth on 100BT ethernet). In Figure 4 we used DPSS servers with the same network throughput and latency, but varied the server CPU power available by using servers with other jobs running simultaneously. The first server had no load, the second had 33%, the third had 50%, and the fourth had 66% load. In Figure 5 we used servers with the same network throughput and load, but with varied latencies, where the first server had a latency of .5 ms, the second and fourth of 40 ms, the third of 2 ms.

Note that the results for the DPSS without load balancing actually decreases starting with the third server in Figure 3, which is on 10 Mbit/sec ethernet, because without load balancing total throughput is constrained by the speed of the slowest server. The same thing happens in Figure 4 and Figure 5. Therefore the total throughput of the system with no load balancing is the speed of the slowest server, times the number of servers. However, by using simple load balancing, the total throughput of the system is the sum each server, a significant improvement. The overall throughput in Figure 5 is less

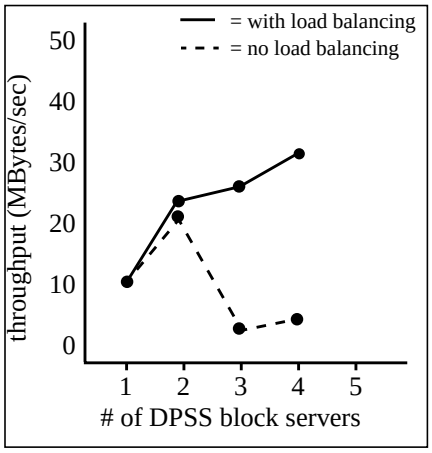

Figure 3: varied network speed

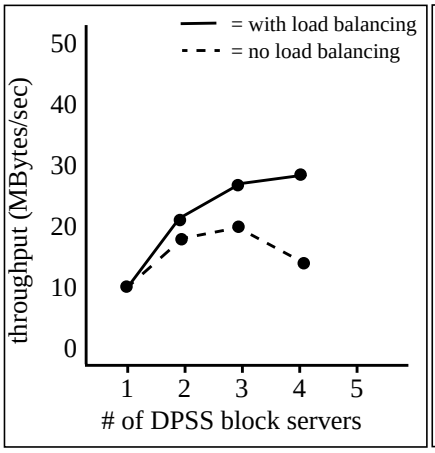

Figure 4: varied server CPU

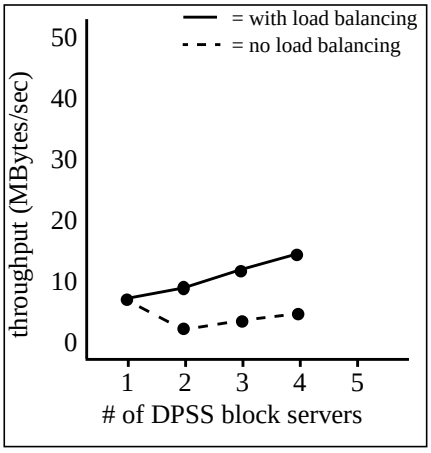

Figure 5: varied latency

than Figures 3 and 4 because the servers used were considerably less powerful, and were connected by networks with a throughput of only 12 Mbits/sec.

Next, we tested all three factors together. We used a DPSS configuration with a wide range of server characteristics, shown in Table 2. With this configuration we obtained the results shown in Figure 6, which shows by doing load balancing using throughput only, overall performance goes up for the two server case, but actually goes down for three or more servers. This is caused by the third server being on a slow and long latency path, and is therefore much slower, and brings down the overall throughput.

Table 2

server	net thruput	CPU	Latency
A	120 Mb/s	1	.5 ms
B	80 Mb/s	.5	1 ms
C	12 Mb/s	2	50 ms
D	12 Mb/s	.5	60 ms

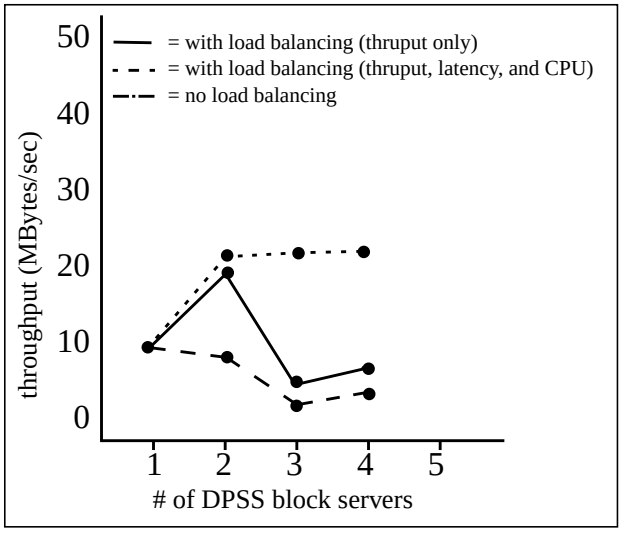

Figure 6: varied CPU, network, and latency

The load balancing formula returns a value of .06% for the third server, effectively disabling that server, and preserving the overall throughput. The same thing happens with the fourth server.

This seems to indicate that the formula we are using does a reasonably good job of load balancing. More experimentation remains to be done to verify the generality of our load balancing formula, and compare it to other approaches. These results will be in the full version of this paper.

6.0 Related Work

Research on automatic TCP receive buffer tuning in the operating system is being done by the Pittsburgh Supercomputer Center [16]. While this is a very nice approach, it will currently only help applications running on hosts with a customized version of the NetBSD operating system. Our approach can be used by any distributed application on any Unix system.

Other network-aware application or middleware projects include the Remulac project [4][17], which is also doing network monitoring and application adaptation based on the current network conditions. This project is focusing on designed a middleware API for applications to use, where we are focused instead on issues for data intensive computing.

7.0 Conclusions

We have shown that by adding network-awareness to a distributed system one can greatly improve performance. We believe that this type of network-aware network cache will be an important architectural component to building effective data intensive computational grids.

8.0 References

[1] Erol Akarsu, Geoffrey C. Fox, Wojtek Furmanski, Tomasz Haupt, "WebFlow - High-Level Programming Environment and Visual Authoring Toolkit for High Performance Distributed Computing", Proceedings of IEEE Supercomputing '98, Nov. 1998.

[2] Chaitanya Baru, Reagan Moore, Arcot Rajasekar, Michael Wan, "The SDSC Storage Resource Broker", Proc. CASCON'98 Conference, Nov.30-Dec.3, 1998, Toronto, Canada. (http://www.npaci.edu/DICE/SRB/)

[3] K. Czajkowski, I. Foster, C., Kesselman, N. Karonis, S. Martin, W. Smith, S. Tuecke. "A Resource Management Architecture for Metacomputing Systems", Proc. IPPS/SPDP '98 Workshop on Job Scheduling Strategies for Parallel Processing,1998.

[4] DeWitt, T. Gross, T. Lowekamp, B. Miller, N. Steenkiste, P. Subhlok, J. Sutherland, D., "ReMoS: A Resource Monitoring System for Network-Aware Applications" Carnegie Mellon School of Computer Science, CMU-CS-97-194. http://www.cs.cmu.edu/afs/cs/project/cmcl/www/remulac/remos.html

[5] DPSS: http://www-didc.lbl.gov/DPSS

[6] Globus: See http://www.globus.org

[7] Grid: *The Grid: Blueprint for a New Computing Infrastructure*, edited by Ian Foster and Carl Kesselman. Morgan Kaufmann, Pub. August 1998. ISBN 1-55860-475-8. http://www.mkp.com/books_catalog/1-55860-475-8.asp

[8] Habanero: http://www.ncsa.uiuc.edu/SDG/Software/Habanero/

[9] V. Jacobson, "Congestion Avoidance and Control", Proceedings of ACM SIGCOMM '88, August 1988.

[10] JAMM: http://www-didc.lbl.gov/JAMM/

[11] William E. Johnston. "Real-Time Widely Distributed Instrumentation Systems," In *The Grid: Blueprint for a New Computing Infrastructure*. Edited by Ian Foster and Carl Kesselman. Morgan Kaufmann, Pubs. August 1998.

[12] William E. Johnston, W. Greiman, G. Hoo, J. Lee, B. Tierney, C. Tull, and D. Olson. "High-Speed Distributed Data Handling for On-Line Instrumentation Systems," Proceedings of ACM/IEEE SC97: High Performance Networking and Computing. Nov., 1997. http://www-itg.lbl.gov/~johnston/papers.html

[13] Legion: See http://www.cs.virginia.edu/~legion/

[14] MAGIC: "The MAGIC Gigabit Network." See: http://www.magic.net

[15] Netperf: http://www.netperf.org/

[16] J. Semke, J. Mahdavi, M. Mathis, "Automatic TCP Buffer Tuning", Computer Communication Review, ACM SIGCOMM, volume 28, number 4, Oct. 1998.

[17] P. Steenkiste, "Adaptation Models for Network-Aware Distributed Computations", 3rd Workshop on Communication, Architecture, and Applications for Network-based Parallel Computing (CANPC'99), Orlando, January, 1999.

[18] Brian Tierney, William E. Johnston, Hanan Herzog, Gary Hoo, Guojun Jin, Jason Lee, Ling Tony Chen, Doron Rotem. "Distributed Parallel Data Storage Systems: A Scalable Approach to High Speed Image Servers," ACM Multimedia '94 (San Francisco, October 1994). http://www-itg.lbl.gov/DPSS/papers/